Tableau: A High-Throughput and Predictable VM Scheduler for High-Density Workloads

Manohar Vanga
MPI-SWS
Kaiserslautern, Germany

Arpan Gujarati
MPI-SWS
Kaiserslautern, Germany

Björn B. Brandenburg
MPI-SWS
Kaiserslautern, Germany

ABSTRACT

In the increasingly competitive public-cloud marketplace, improving the efficiency of data centers is a major concern. One way to improve efficiency is to consolidate as many VMs onto as few physical cores as possible, provided that performance expectations are not violated. However, as a prerequisite for increased VM densities, the hypervisor's VM scheduler must allocate processor time efficiently and in a timely fashion. As we show in this paper, contemporary VM schedulers leave substantial room for improvements in both regards when facing challenging high-VM-density workloads that frequently trigger the VM scheduler. As root causes, we identify **(i)** high runtime overheads and **(ii)** unpredictable scheduling heuristics. To better support high VM densities, we propose Tableau, a VM scheduler that *guarantees* a minimum processor share and a maximum bound on scheduling delay for every VM in the system. Tableau combines a low-overhead, core-local, table-driven dispatcher with a fast on-demand table-generation procedure (triggered on VM creation/teardown) that employs scheduling techniques typically used in hard real-time systems. In an evaluation of Tableau and three current Xen schedulers on a 16-core Intel Xeon machine, Tableau is shown to improve tail latency (*e.g.*, a 17× reduction in maximum ping latency compared to Credit) and throughput (*e.g.*, 1.6× peak web server throughput compared to RTDS when serving 1 KiB files with a 100 ms SLA).

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; **Real-time systems**; • **Software and its engineering → Virtual machines**; **Scheduling**; Real-time schedulability;

KEYWORDS

Virtualization, Hypervisor Scheduling, Real-Time Scheduling

ACM Reference Format:
Manohar Vanga, Arpan Gujarati, and Björn B. Brandenburg. 2018. Tableau: A High-Throughput and Predictable VM Scheduler for High-Density Workloads. In *EuroSys '18: Thirteenth EuroSys Conference 2018, April 23–26, 2018, Porto, Portugal.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3190508.3190557

EuroSys '18, April 23–26, 2018, Porto, Portugal

ACM ISBN 978-1-4503-5584-1/18/04…$15.00
https://doi.org/10.1145/3190508.3190557

1 INTRODUCTION

As the marketplace for public clouds matures and cloud services are being increasingly commoditized, cloud providers are forced to continuously increase the efficiency of their data centers, and to improve the price/performance ratio of their various service tiers, especially at the low end.

One way to increase data center efficiency is to pack a growing number of VMs onto fewer physical cores. This reduces resource wastage as active cores serving multiple lower-tier VMs are highly utilized while the number of paying customers relative to the required infrastructure can be increased. Alternatively, any freed-up machines can be used to support higher-tier (and higher-priced) VMs that require dedicated processing cores. Either way, the ability to consolidate a larger number of lower-tier VMs onto fewer cores—*i.e.*, the ability to pack VMs as tightly as possible without violating customer expectations—is a distinct economic advantage in the competitive cloud marketspace. However, consolidating VMs onto shared cores is easier said than done as customers desire high throughput and reasonably low and stable latency characteristics even for lower-tier VMs.

A key hypervisor component that affects these central metrics—application throughput and latency, as perceived by the customer—is the *VM scheduler*. In particular, if the VM scheduler is inefficient (*i.e.*, if it suffers from large runtime overheads), then the peak throughput attainable by guest VMs will be needlessly limited. Furthermore, while application tail latency is a complex phenomenon that is determined by multiple factors, if the VM scheduler occasionally *induces* a substantial amount of *scheduling latency* due to poor scheduling decisions, then application tail latency will inevitably suffer. In other words, if the VM scheduler is a major bottleneck, then it will surely impose a tax on application performance.

Unfortunately, many of the VM schedulers in widespread use today, and especially those used in Xen, are not yet optimized for hosting highly consolidated, high-VM-density workloads, and have little to offer in terms of performance *guarantees*. In particular, as we show in this paper (Sec. 7), Xen's existing schedulers can negatively affect either tail latencies, throughput, or both due to the use of unpredictable scheduling heuristics that sometimes backfire and/or implementation aspects that cause undesirably high overheads, especially when faced with a large number of densely packed VMs.

Motivated by these observations, this paper presents *Tableau*, a highly predictable, high-throughput VM scheduler based on an unorthodox design not previously explored in a data-center context. Specifically, Tableau leverages multiprocessor scheduling techniques typically used in hard real-time systems, and exploits specific properties of cloud environments to minimize runtime overheads, unlike prior real-time VM schedulers (*e.g.*, RT-Xen [74]).

Tableau consists of two main components: **(i)** a low-overhead, table-driven, core-local dispatcher that schedules VMs primarily based on a given static schedule, and **(ii)** an asynchronous, infrequently invoked planner that re-generates tables on-demand when VMs are either created, torn down, or reconfigured.

As a result of this clear separation between a semi-offline planning phase and an extremely simple online dispatcher, Tableau incurs significantly lower runtime overheads (around 5.6×, 2.4×, and 2× lower than under Credit, Credit2, and RTDS, respectively, see Sec. 7.2). These efficiency gains in turn can translate into substantial improvements in SLA-aware throughput (*e.g.*, compared to RTDS, Tableau can achieve up to 1.6× higher peak throughput when serving 1 KiB files with a 100 ms SLA, see Sec. 7.4).

Furthermore, Tableau's inherent predictability can yield substantially improved tail latency characteristics for workloads that frequently invoke the VM scheduler (*e.g.*, in some cases VMs scheduled by Tableau exhibit up to 17× lower maximum ping latency compared to Credit, the default Xen scheduler, see Sec. 7.3).

Key to Tableau is the planning stage, which is performed asynchronously. It thus only affects the creation, teardown, and reconfiguration time of VMs (inflating each one by a few hundred milliseconds), an acceptable tradeoff for relatively infrequent operations that usually take on the order of seconds to begin with.

Each VM under Tableau is configured with a minimum CPU budget (or *utilization*) and a maximum-acceptable scheduling delay, both of which can be determined either based on an explicit SLA, based on pre-determined, price-differentiated service tiers offered by cloud vendors, or empirically based on the deployed workload or simple fair-share policies. Tableau's planner applies techniques from hard real-time multiprocessor scheduling to *quickly* re-generate scheduling tables whenever needed, while ensuring that all constraints on the minimum utilization and maximum scheduling latency for every VM in the system are satisfied. Consequently, Tableau provides direct control over one of the key contributors to tail latencies, namely the scheduling latency of individual VMs.

Contributions. We present the design of Tableau (Sec. 3), an unorthodox scheduling approach rooted in static scheduling tables (as pioneered in hard real-time systems [35]), which has not previously been explored in a cloud context. Tableau requires on-demand generation of scheduling tables satisfying the utilization and scheduling-latency constraints of individual vCPUs. We detail how to quickly find such tables by repurposing relevant real-time scheduling theory (Sec. 5), and report on an efficient implementation of Tableau in Xen 4.9 (Sec. 6). Notably, our implementation is inherently scalable because it uses almost exclusively core-local data structures. In an evaluation with an I/O-intensive workload on a dual-socket, 16-core Intel Xeon platform (Sec. 7), our Tableau prototype is shown to outperform the existing Xen schedulers (RTDS, Credit, and Credit2) in terms of their SLA-aware peak throughput.

In summary, Tableau is a novel, predictable, high-throughput VM scheduler designed to ensure that VM scheduling is no longer a bottleneck in the support for high-density VM workloads.

2 THE PROBLEM

Each VM consists of one or more vCPUs. The role of the VM scheduler is to multiplex the vCPUs of all VMs onto the available physical CPU cores (pCPUs) such that each VM receives (at least) a certain configured share of CPU time while minimizing the latency and maximizing the throughput exhibited by tenant applications.

As we explain next, these goals are affected by **(i)** *unpredictable scheduling heuristics*, as employed by many popular VM schedulers, as they may cause increased tail latencies, and **(ii)** *high runtime overheads*, which eat up precious cycles and thus lower throughput.

2.1 Unpredictability Increases Tail Latencies

To improve the average-case latency of VMs, many VM schedulers contain heuristics and special-case optimizations that favor VMs performing I/O [17, 52, 67].

For example, Xen's default *Credit* scheduler, a design rooted in the principle of proportionate fairness [31], "boosts" the priority of vCPUs that resume from a blocking I/O operation to temporarily override the fairness criterion. Similarly, Linux's *Completely Fair Scheduler* (CFS), which is widely used in conjunction with Linux's built-in *KVM* hypervisor, also uses accounting tricks to favor I/O activity (*e.g.*, the "gentle fair sleepers" setting). In fact, CFS has even been observed to under-utilize cores in fully loaded systems due to complex and erratic (and erroneous) load-balancing heuristics [43].

Unfortunately, the effects of such heuristics are difficult to anticipate and can result in increased tail latencies. For instance, whether Xen's "boosting" heuristic actually reduces I/O latency depends on the number of simultaneously boosted vCPUs: if every vCPU is performing I/O and boosted as a result, then effectively no vCPU is boosted. As we show in our evaluation, such runtime heuristics increase performance volatility, which translates into increased, scheduler-induced tail latencies for high-VM-density workloads.

2.2 Scheduling Overhead Limits Throughput

A second major concern is that a VM scheduler must exhibit very low runtime overheads, because it can be frequently invoked, and because any cycles spent on scheduling are pure overhead in that they would otherwise have been available to applications of paying customers. Thus, in addition to causing unpredictable latency spikes, dynamic scheduling heuristics can also be detrimental to throughput because they must be frequently computed. Similarly, any other major source of runtime overheads such as lock contention inside the scheduler hurts application throughput.

At cloud scale, such overheads can add up to massive costs. For example, a recent study of more than 20,000 computers in one of Google's data centers found that roughly 5% of all processor cycles are spent on the kernel's process scheduler [30]. In the case of high-density workloads, the effects of any scheduling bottlenecks are further exacerbated by their naturally higher context-switch rates.

There is thus strong motivation to make the VM scheduler as efficient as possible. However, as we demonstrate in our evaluation, contemporary VM schedulers leave substantial room for improvements in terms of both runtime overheads and scheduler-induced tail latencies when facing challenging high-VM-density workloads that frequently trigger the VM scheduler. As a novel, unorthodox alternative that occupies a previously unexplored point in the design space of VM schedulers, we propose Tableau, a low-overhead VM scheduler that *guarantees* a minimum share of CPU time and a hard bound on maximum scheduling latency for every vCPU.

3 THE TABLEAU APPROACH

Tableau is based on a table-driven design inspired by hard real-time systems to minimize runtime overheads while maintaining high throughput and predictable latencies even when confronted with a large number of VMs. In the following, we introduce the high-level design; implementation-level optimizations are discussed in Sec. 6.

Dispatcher vs. planner. VMs in cloud environments are typically long-running. For example, a majority of customer VMs hosted on Microsoft Azure have a lifetime of at least a few hours [19], and VMs that run longer than a day are likely to run for several days and account for more than 95% of the total core hours [19]. Based on this observation, we push all expensive scheduling logic related to satisfying VM performance requirements into a separate, infrequent planning (or *system reconfiguration*) step that is only invoked when a VM is started up, torn down, or reconfigured.

Consequently, Tableau's scheduler consists of two main components: a straightforward, low-overhead, table-driven *dispatcher* and a relatively heavy-weight *scheduling table generator* (or *planner*). The dispatcher resides in the hypervisor and is invoked whenever a scheduling decision is needed. It simply enacts the latest scheduling table provided by the planner. The planner in turn can reside anywhere (*e.g.*, it can be an unprivileged process) and is invoked if a new table is needed, *i.e.*, when a system reconfiguration occurs. By leveraging multiprocessor real-time scheduling theory (discussed in Sec. 5), the planner *quickly* generates tables that *guarantee* a minimum CPU share and a hard upper bound on the maximum scheduling delay for each VM in the system.

An immediate benefit of the split between a minimal, efficient dispatcher and a separate system-wide planning process is that the dispatcher uses *primarily core-local data structures*, which trivially ensures Tableau's scalability on large multicore platforms. All decisions requiring system-wide information and coordination as well as table updates are done asynchronously by the planner and do not slow down the online dispatcher. Thus, the performance-critical scheduler hot paths are not impacted by the planner's overheads.

Second-level scheduler. A naive table-driven scheduler, however, is too inflexible at runtime and results in non-work-conserving behavior. Since it is generally the goal of cloud operators to use each and every last cycle to maximize profits, this is clearly not an acceptable limitation. In Tableau, we hence overcome this limitation by incorporating a simple second-level round-robin fair-share scheduler that is invoked whenever the first-level table-driven scheduler fails to find a vCPU to schedule.

To summarize, Tableau is a two-level, hierarchical scheduling approach with a table-driven dispatcher at the first level, a core-local fair-share scheduler at the second level, and an infrequently invoked asynchronous planner. Together, these components ensure flexible, work-conserving runtime behavior on top of the minimum performance guarantees incorporated into the tables, which are (re-)generated on demand.

We next elaborate on the dispatcher and then discuss how the planner finds scheduling tables with performance guarantees in Sec. 5.

4 A MINIMAL DISPATCHER

At the first level, Tableau schedules vCPUs using a table-driven dispatcher, not unlike those commonly found in safety-critical hard real-time systems. For instance, the ARINC 653 standard for *integrated modular avionics* [56] specifies time-partitioned scheduling, which is accomplished with static scheduling tables. We adopt this proven technique for building highly predictable systems.

A table-driven dispatcher requires a pre-generated scheduling table of finite length, usually in the range of a few hundred milliseconds. For each core, the table is given as a set of non-overlapping intervals, each of which is specified using offsets relative to the start of the table. Each interval is either marked as idle or reserved for a specific vCPU that is given priority during the interval.

When the dispatcher is invoked at runtime, it simply looks up the interval in the scheduling table covering the current time (modulo the table length). If this interval is reserved for a specific vCPU, and if that vCPU is ready, it is dispatched and allowed to run uninterruptedly until the end of the interval. If the specific vCPU is blocked, or if the interval is marked as idle, the second-level scheduler is invoked to schedule any ready *core-local* vCPU, which is chosen based on additional information provided in the tables. The second-level scheduler is a fair-share, epoch-based scheduler: it divides the time within each (configurable) epoch evenly among the runnable vCPUs into per-vCPU budgets and then applies a simple highest-remaining-budget-first policy. Second-level vCPU budgets are replenished when all ready vCPUs have run out of budget.

To summarize, the scheduler hot path in Tableau consists of little more than a straightforward table lookup in the common case, which is a minimal and hence extremely efficient approach to scheduling. The schedule resulting from the table repeats cyclically until a new table is installed by the planner. Importantly, it is inherently predictable: the maximum "blackout time" during which a vCPU receives no service, and which directly translates into application-visible latency, is trivially bounded. It is also work-conserving (w.r.t. core-local vCPUs) owing to the second-level scheduler.

To be clear, the design of Tableau is intentionally simple: our claim is *not* that Tableau is a particularly sophisticated approach, but rather that such a simple, largely static design not only suffices to serve cloud workloads, but that it can actually exceed the performance of those widely used today, as demonstrated in Sec. 7.

Next, we introduce the actual scheduling logic (*i.e.*, how the planner generates scheduling tables) and then discuss a concrete realization of Tableau in Xen in Sec. 6.

5 FINDING A GOOD SCHEDULE QUICKLY

Each VM comprises one or more vCPUs. As input, the planner requires a specified *reserved utilization* U and a *maximum scheduling latency* L for each vCPU. These parameters may be selected arbitrarily. For instance, they can be explicitly specified by an associated SLA, pre-determined according to price-differentiated service tiers set by the cloud provider, or simply computed by a fair-share policy (*e.g.*, $U = \frac{m}{n}$, where m is the number of CPU cores and n the number of vCPUs assigned to the host).[1] The challenge is

[1] In particular, note that Tableau does *not* require more information to be provided than existing fair-share schedulers such as Xen's Credit scheduler or Linux's CFS scheduler—just as in Credit or CFS, U can be determined automatically based on a vCPU- or VM-specific weight, the number of cores, and the number of vCPUs in the system, and L can be given a reasonable default magnitude similar to the scheduling quantum in Credit or the `sched_latency_ns` tunable of CFS. In *addition*, Tableau allows for more sophisticated or price-differentiated provisioning strategies, but *not* at the price of a more complicated default setup or a higher barrier to adoption.

to find a static vCPU schedule for the dispatcher that is runtime-efficient and that satisfies the minimum utilization and maximum latency guarantees for all vCPUs.

Since the planner can operate outside the restricted confines of the hypervisor, such as within a supervisory VM, one might be tempted to use high-level tools such as ILP or SMT solvers to find schedules. However, we want the table generation process to be relatively fast (*i.e.*, seconds rather than minutes) even for hundreds of vCPUs and therefore avoid such heavyweight solutions.

We thus instead map the problem to the well-studied multiprocessor hard real-time scheduling problem, which allows us to quickly generate reasonably short tables satisfying all constraints for *any possible configuration* of VMs that does not over-utilize the system (*i.e.*, where the sum of all U parameters does not exceed the number of available cores).

Specifically, we generate tables in two steps. We first model each vCPU as a *periodic task* [41] with parameters that are carefully chosen to **(i)** reflect its specified U and L parameters while **(ii)** ensuring a short maximum table length. We then simulate a multiprocessor real-time schedule of the set of periodic tasks representing all vCPUs in the system. This simulation results in a repeating table that achieves the target utilization and ensures the desired scheduling latency for each vCPU.

Mapping to periodic tasks. A periodic real-time task [41] $\tau = (C, T)$ is characterized by its *worst-case execution time* C and *period* T. The associated correctness criterion is that it must receive (up to) C time units of processor service in each scheduling interval $[0, T)$, $[T, 2T)$, $[2T, 3T)$, *etc.*

When mapping a vCPU (U, L) to a periodic task $\tau = (C, T)$, we clearly require $U = \frac{C}{T}$. However, how does a vCPU's latency goal L map to an "equivalent" period T?

Without knowing anything about the final schedule, a suitable period can be determined by observing that a periodic task must be scheduled for at least C time units during every period of length T. The worst-case blackout time (*i.e.*, contiguous interval without any processor service) hence occurs when a periodic task is scheduled for C time units at the very beginning of one period, and then scheduled next only at the very end of the next period, again for C time units. For example, a periodic task with $(C, T) = (10ms, 100ms)$ might be scheduled during $[0ms, 10ms)$ and then again during $[190ms, 200ms)$, yielding a blackout time of $180ms$ corresponding to the blackout interval $[10ms, 190ms)$.

In general, the worst-case blackout time incurred by a periodic task with period T and cost C is bounded by $2 \times (T - C)$, or equivalently $2 \times (1 - U) \times T$. Thus, a vCPU's latency goal L can be converted into a suitable period T by picking any period T such that $T \leq \frac{L}{2 \times (1-U)}$. (If $U = 1$, then the vCPU is simply mapped to a dedicated pCPU and excluded from further consideration.)

Bounding table lengths. To minimize preemptions, one should maximize the period. However, simply choosing the maximal period for each vCPU can result in a periodic task set with an extremely large *hyperperiod*, the least common multiple of all task periods. Since this is also the length at which the dispatching table repeats, picking periods indiscriminately could even result in exponential table sizes (if all chosen periods are relatively prime).

To avoid large dispatching tables, we select periods from a set of candidate periods with a known maximum hyperperiod. Specifically, in our implementation, we searched for a number close to 100 ms (=100,000,000 ns) that has a large number of factors larger than $100\mu s$ (since periods smaller than $100\mu s$ are hard to enforce due to scheduling overheads). We chose 102,702,600 ns as the maximum hyperperiod, which has a large number of integer divisors (186) above the $100\mu s$ threshold. The length of approximately $102\,ms$ is short enough to be generated and replaced quickly.

Thus, if F denotes the set of all integer divisors of 102,702,600 greater than 100,000, we select for each vCPU the largest $T \in F$ such that $2 \times (1 - U) \times T \leq L$. Depending on the chosen T, tenants may observe less scheduling delay than stipulated by L, which is consistent with it being an *upper bound* on scheduling latency.

Once each vCPU is represented as a periodic task, the planner must find a schedule that satisfies the timing constraints of all periodic tasks, in which case all vCPU utilization and latency goals are guaranteed to be met. To this end, Tableau uses a progression of three increasingly expensive techniques: first a very simple and quick bin-packing heuristic that we expect to be sufficient in most practical use cases, and then two more involved scheduling techniques that we include primarily for the sake of completeness (*i.e.*, to ensure that the planner never fails, even in pathological scenarios).

Partitioning. We begin by attempting to *partition* the task set (*i.e.*, statically assign tasks to cores) such that no core is overloaded. Such an approach is a desirable first step as it results in high cache affinity (since no vCPUs migrate between cores). Partitioning also has the advantage that additional considerations such as memory locality on NUMA platforms, special treatment of hardware threads, or cache interference concerns can be easily incorporated.

Partitioning periodic tasks onto cores is a bin-packing-like problem that is NP-hard. We use the well-known *worst-fit decreasing* heuristic (always assign the next task to the least-utilized core), which has the benefit that it distributes the load roughly evenly across all cores in the system.

If the partitioning heuristic succeeds in finding a valid partition, we simply simulate on each core an *earliest-deadline-first* (EDF) schedule until the hyperperiod. Since EDF is optimal on uniprocessors [41], the simulation guarantees a schedule satisfying all utilization and latency goals.

It bears repeating that we expect this partitioning step to succeed in most cases in practice. This is particularly true in the context of cloud data centers: since the cloud provider controls the dimensioning of the various service tiers, it can arrange for a suitably simple bin-packing problem by offering only regularly sized VMs.

If partitioning fails, however, we attempt *semi-partitioning* [4].

Semi-partitioning. Semi-partitioning is a simple extension of partitioning. First, we try to partition the task set as before. However, when encountering a task that cannot be assigned to any core, instead of giving up, the task is broken up into smaller *subtasks* with precedence constraints, which are then easier to partition. The subtasks represent the task's fractional allocations on different cores. At runtime, a split task migrates among the cores to which its subtasks have been assigned to use the reserved processor time.

The trick is to ensure **(i)** that the subtasks never execute in parallel (since they still reflect the same sequential task), and **(ii)** that

no core becomes overloaded. In general, this is not trivial, but many suitable semi-partitioning schemes have been proposed in recent years [5, 6, 9, 10, 12, 32, 33, 39].

Without going into too much detail, we simply apply one proposal called *C=D task-splitting* [12] that virtually always finds a valid split, even for difficult problem instances that almost fully utilize all cores [11]. Finding valid *C=D* task splits is non-trivial, coNP-hard [22], and computationally demanding in general; however, due to the fixed table length, it is fast in Tableau's use case.

If semi-partitioning succeeds, the planner again simulates an EDF schedule on each core.

Localized optimal scheduling. While the *C=D* approach is empirically near-optimal [11], *i.e.*, it is virtually always possible to find workable task splits, there nonetheless exists theoretically a chance that it might fail. In such a case, which we never encountered in our evaluation, it is possible to fall back to *optimal* multiprocessor real-time scheduling as a last resort [8, 25, 48, 59, 63]. Although optimal schedulers guarantee the existence of a schedule, we do not use them as our first choice since they tend to generate many preemptions and migrations. Instead, we perform semi-partitioning to the extent possible, and use an optimal scheduler to schedule the remaining tasks on a minimal subset of cores.

Specifically, we identify two physical cores that are "close" (*e.g.*, that share a cache) and turn them into a *cluster* (*i.e.*, a "double-sized bin") that is optimally scheduled. This merging of bins is repeated if needed until all tasks can be partitioned, split into subtasks, or assigned to some cluster of cores. The process is guaranteed to stop when reaching a single cluster encompassing all cores (if the system is not over-utilized). However, we emphasize that this procedure is virtually never needed for practical workloads; we include it simply so that table generation truly never fails (unless the system is over-utilized, which is a misconfiguration that is rejected).

It is worth mentioning that vCPUs that migrate among two or more pCPUs due to semi-partitioning (or localized optimal scheduling) represent a complication for the second-level scheduler—on which pCPU should such a vCPU participate in the second-level scheduling? To avoid costly synchronization, one straightforward approach is to adopt a "trailing core" policy: migrating vCPUs participate in the second-level schedule (only) on the pCPU on which they last received a guaranteed allocation.

Post-processing. After a schedule has been found, the planner performs certain post-processing operations before handing the schedule over to the dispatcher. First, it coalesces allocations below a certain threshold into a neighboring allocation. This threshold is determined by the overheads involved in context-switching vCPUs, since allocations smaller than the threshold are impossible to enforce. In the last step, the planner "slices" the table to enable constant-time lookups, as discussed in the following section.

Finally, while we have not explored this space yet, it is trivial to add additional post-processing steps. For instance, one might add a "peep-hole" optimization pass to reduce the number of migrations and preemptions even further. Alternatively, one might add a pass to encourage or discourage co-scheduling of certain VMs, *e.g.*, due to performance-counter-based profiles or for synchronization purposes. We leave these interesting opportunities and extensions to future work.

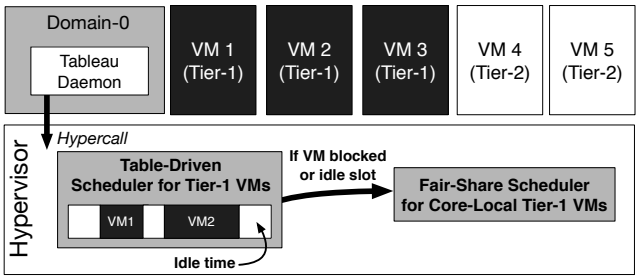


Figure 1: Tableau architecture

6 IMPLEMENTATION: TABLEAU IN XEN

The Tableau approach is not tied to any particular system and can be realized in virtually any modern hypervisor. For evaluation purposes, we chose the popular Xen hypervisor (version 4.9) as the basis for our experiments, due to its widespread use in public clouds.

The main components of Tableau in Xen are illustrated in Fig. 1. Xen consists of a special supervisory VM called *domain-0*, or *dom0*, which has privileged access to the underlying hardware to enable **(i)** device access, and **(ii)** the creation, teardown, and reconfiguration of domains. Accordingly, the planner is realized as a daemon in the userspace of dom0 (henceforth referred to simply as userspace).

Implementing the scheduling logic in userspace is quite convenient. In particular, the Tableau planner is written in Python using *SchedCAT*, an open-source real-time scheduling toolkit [1]. The use of a high-level language greatly simplifies the rapid exploration of new post-processing phases and scheduling ideas, potentially even by non-systems developers or using machine-learning techniques.

In total, our Tableau prototype consists of around 2,350 lines of new or changed C code in the hypervisor itself, and around 1,600 lines of code in the userspace Tableau daemon. It was possible to realize Tableau with a relatively small code base because the hypervisor component is simple by design, and because the planner heavily relies on existing scheduling logic [11] in SchedCAT.

New scheduling tables are pushed by the planner to the hypervisor via a hypercall in a compiled, binary format and used directly by the Tableau dispatcher. While the dispatcher is conceptually straightforward, there are certain choices involved in implementing it efficiently. In the following, we highlight four key aspects.

O(1) dispatch. Fig. 2 shows the structure of a Tableau scheduling table. It consists of per-CPU lists of allocations, which map an interval of time to a specific vCPU.

An allocation represents a variable-length interval within the table. To facilitate constant-time lookups, the scheduling table is accompanied by a *slice table*. A slice table is comprised of "slices" of the allocation table, where each slice describes a *fixed-sized time interval* of the allocation table. The slice length is chosen such that each slice overlaps with at most two allocations (and possibly some idle time between them). This is accomplished by picking, for each pCPU, a per-CPU slice length equal to the length of the shortest allocation on that particular pCPU.

The slice table enables $O(1)$ scheduling decisions. First, the dispatcher determines the current slice by indexing the slice table using the current time (modulo the table length), and then it determines which of the two allocations within the slice (or the idle

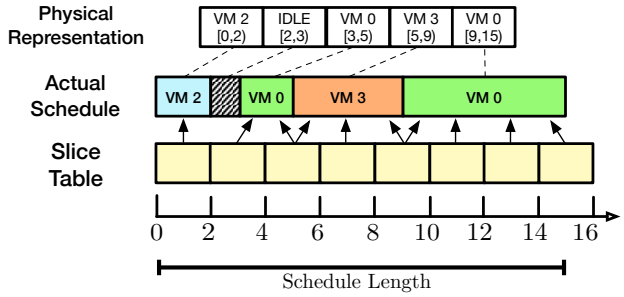


Figure 2: Tableau table structure. The lookup table enables constant-time lookups into the physical table structures using time offsets relative to the start of the table.

time between them) currently need to be scheduled. Allocation and slice records are aligned to cache lines, so at most two cache lines are accessed per lookup.

The simple table-driven dispatcher is efficient and inherently scalable as most memory accesses are to core-local data structures, especially in the common-case hot path. However, in two exceptions, which we briefly sketch next, remote accesses are needed.

Cross-core migrations. When a vCPU has allocations on two cores (*e.g.*, due to semi-partitioning), and if the gap between these two allocations in the table is small (or even overlapping by a few cycles due to timer skew), we must ensure that one core does not schedule the vCPU until it has been completely de-scheduled on the other core (to avoid stack corruption).

To this end, for each vCPU, Tableau tracks the core that currently schedules the vCPU, if any. Before scheduling a vCPU, a core checks that it "owns" the vCPU. If the vCPU is still marked as "scheduled elsewhere," the core that failed to schedule the vCPU sets a field in the vCPU structure requesting an *inter-processor interrupt* (IPI) to be sent when the vCPU is de-scheduled, and schedules either a vCPU selected by the second-level scheduler or idles until notified. No locks or cache lines shared by all pCPUs are required.

In the expected case (no overlap of allocations), the only cost is an atomic write to the vCPU control block (which is already in cache anyway). In the rare case of a race between allocation start and end times, a remote memory reference and an IPI are occasionally incurred, which however does not impact scalability.

Efficient wake-ups. Tableau must also deal with wake-ups of blocked vCPUs, which may be processed on any core in the system. For each vCPU, we keep track of the core it currently has an allocation on (or where it last had an allocation). When a core processes a wake-up for a vCPU that has a current allocation, it reads this field and sends an IPI to the responsible core. Similarly, if the vCPU does not have a current allocation, but is allowed to take part in second-level scheduling, and the vCPU's last-used core is currently idling, then an IPI is sent to said core.

If, however, the vCPU does not currently have an allocation on any core and is capped (*i.e.*, not eligible to take part in second-level scheduling), then the wake-up can be safely ignored; when the next allocation pertaining to the vCPU begins, it will be seen to be runnable anyway. Again, no locks or globally shared cache lines are required to realize this optimization.

For simplicity, it is also possible to unconditionally send an IPI; if IPIs are relatively cheap, then this may be preferable to complicating the wake-up logic. Our prototype currently uses this approach.

Lock-free table switches. Finally, to avoid adding a lock or a barrier in a hot path, table switches in Tableau are time-synchronized. Each core has a *next_table* pointer, which is set when a new table is pushed. If a core finds this field to be set when the current table wraps around, then it switches to the new table (otherwise the current one is reused). However, if the *next_table* pointer is set *during* a table wrap, some cores may pick up the change while others may retain the old table, causing an inconsistent schedule. To avoid such races, we simply ensure that tables are never set during or close to a table wrap. When a new table is pushed, all *next_table* pointers are timed to be set at a point in the middle of the next round of the current table. Given the scheduling table length in Tableau ($\approx 102\,ms$), this technique avoids any race and all cores consistently switch to the new table. Two rounds after a new table has been uploaded, when all cores have switched to the new table, the previous table is garbage-collected.

In this paper, our evaluation is focused on partitioning, and so we did not implement second-level scheduling for semi-partitioned VMs, which means they cannot currently make use of spare idle cycles. However, this feature is trivial to incorporate into Tableau as the minimal synchronization needed for cross-core migrations already exists, as described earlier, and thus its absence is not a major limitation. Our implementation is available online.[2]

7 EVALUATION

Our evaluation is aimed at validating the following key claims: **(i)** Tableau incurs low scheduling overheads compared to other Xen schedulers; **(ii)** it offers both predictability (*i.e.*, consistent, low latencies) and high throughput in a high-VM-density scenario; and **(iii)** planning overheads are acceptable relative to typical VM commissioning and decommissioning times. We start with the latter.

7.1 Table-Generation Overheads

The time and memory overhead of Tableau's planner varies depending on **(i)** the number of VMs, and **(ii)** the configuration of individual VMs, and **(iii)** the number of cores in the system. Together, these parameters determine the number of slots and slices that need to be generated, optimized, and written to disk.

To show how these choices affect Tableau's planner, we measured both the time taken to generate tables, as well as the size of the generated tables for a varying number of VMs, with all VMs being assigned one of four latency goals (1ms, 30ms, 60ms, and 100ms). To stress the planner and test its scalability limits, we performed these experiments on and for a 48-core Intel Xeon (E7-8857) server, the largest machine in our lab at the time of writing. Four cores were dedicated to dom0, and four guest VMs were admitted for each of the remaining forty-four cores.

Table-generation time. In Fig. 3, the Y axis shows the total time taken to generate the table (averaged over 100 runs) as a function of the number of VMs for which the table was generated. The number of VMs was varied up to a total of 176 VMs (*i.e.*, four VMs per core).

[2] https://people.mpi-sws.org/~bbb/papers/details/eurosys18/

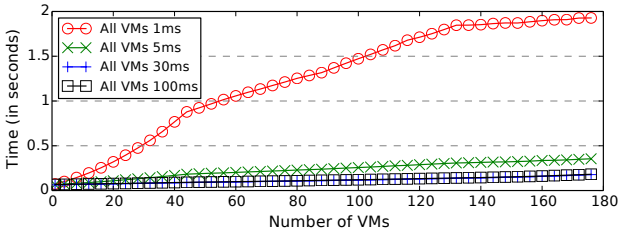


Figure 3: Table-generation times for a varying number of VMs with different latency goals. The 30 ms and 100 ms curves overlap.

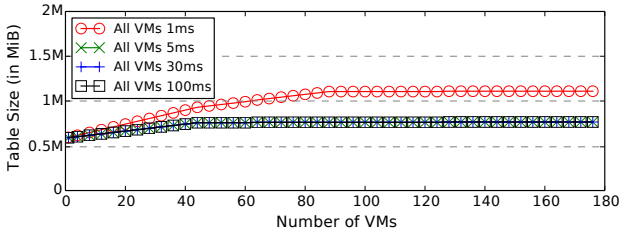


Figure 4: Generated table size for a varying number of VMs with different latency goals. All but the 1 ms curve overlap.

As can be seen in the figure, table-generation time never exceeds two seconds. We believe this to be acceptable in the context of public clouds where typical VM lifetimes far outweigh VM startup, teardown, and reconfiguration times [19]. In contexts where system re-configuration time may be crucial (*e.g.*, Tableau-based container scheduling, or even high-priority process scheduling), several optimizations could be made: **(i)** tables can be incrementally re-computed on a per-core basis, and **(ii)** a low-level language such as C can be used to reduce language runtime overhead. Furthermore, with Tableau, the planner does not necessarily have to reside on the same machine, *i.e.*, table generation may also be offloaded to a faster, independent machine, similarly to how jobs are scheduled across data centers [60], and it is trivially possible to centrally cache tables for common configurations that are frequently reused.

Additional reconfiguration delays. If the planner resides locally in dom0, where it is triggered on-demand, then Tableau introduces a planning delay to VM startup, teardown, and reconfiguration operations. However, we emphasize that the planning overhead does not affect the performance of VMs once they have commenced execution (*i.e.*, it increases only their provisioning time). As VM creation under Xen already takes many seconds, even without accounting for the time it takes the guest OS to actually boot up (nor any time spend on fetching a VM image from remote storage), we deem even the longest table-generation delay reported in Fig. 3, which is two seconds, to still be acceptable.

Memory overheads. Fig. 4 shows the table size (in MiB) on the Y axis, as a function of the number of VMs on the X axis. The four curves show the table size when all VMs are assigned the same latency goal of 1ms, 30ms, 60ms, and 100ms, respectively.

As can be seen in the figure, the memory overhead for all configurations was below 1.2 MiB, which only occurs for a fairly demanding case of every VM having a latency goal of 1 ms. We consider this to be a negligible overhead for modern server-class machines with hundreds of gigabytes, and even terabytes, of RAM.

7.2 Scheduler Runtime Overheads

We now present microbenchmarks comparing Tableau's scheduler overheads with three different schedulers in Xen.

Platform. We used a 16-core, 3.2 GHz Intel Xeon (E5-2667) server (comprising two sockets with eight cores each) with 512 GiB of RAM, running Ubuntu 16.04.3 LTS (Linux kernel version 4.4.0) on Xen 4.9. We employed an identical client machine, connected on the same network via 10 Gbit/s Ethernet, as a load generator. We disabled all CPU power-saving features for our evaluation to avoid performance unpredictability.

Schedulers. We compared our implementation of Tableau with three stock schedulers in Xen (Credit, Credit2, and RTDS). Credit is the default scheduler in Xen and is a weighted proportionate-fair-share scheduler. That is, each VM is allocated credits proportional to a configured weight, which it "burns" when it executes. Additionally, Credit gives VMs that wake up from an I/O operation a "boost" in priority. Xen's more recent Credit2 scheduler extends the original Credit design with the goal of improving responsiveness, and does this primarily by eliminating Credit's priority boosting as it is now understood to cause performance unpredictability. RTDS, another recent addition to Xen, is a real-time scheduler that, like Tableau, is also based on the periodic task model [41]. However, in contrast to Tableau, and similar to Credit, RTDS is a dynamic scheduler (*i.e.*, it makes all decisions online) based on an EDF policy. RTDS is an interesting baseline to compare against because it provides similar capabilities in terms of predictable control over latency and utilization, while representing an entirely different set of tradeoffs due to its dynamic nature.

Scheduler setup. Due to the number of tunable parameters in each of the evaluated schedulers and the resulting vast configuration space, we did not attempt to exhaustively evaluate every possible parameter combination. Rather, our evaluation is based on a single setup that is intended to be representative of the kind of workloads Tableau is designed to support.

Specifically, on our 16-core server, we assigned four single-vCPU VMs per core (*i.e.*, each with 25% CPU utilization), with four cores dedicated to dom0. Credit was configured according to documented best practices. In particular, we used a global timeslice of 5 ms under Credit as the default 30 ms value is known to be non-ideal for I/O workloads [18]. Under Tableau, to allow for a reasonably fair comparison with Credit, we chose a maximum scheduling latency of 20 ms since Credit with a 5 ms timeslice will, in the presence of four VMs per core, replenish all credits roughly once every 20 ms. This results in the planner picking a period of roughly 13 ms with a budget of about 3.2 ms. To enable a direct comparison, RTDS was configured to match the parameters of Tableau.

Due to differences in capabilities of the various schedulers, we evaluate two distinct scenarios: a "capped" scenario, where VMs are configured with CPU-usage upper bounds (supported by Credit, RTDS, and Tableau), and an "uncapped" scenario, where a VM's

CPU usage is not bounded (supported by Credit, Credit2, and Tableau). We used Ubuntu 16.04.3 LTS as the guest OS.

Overhead results. Under each scheduler, we traced the runtime cost of key scheduling operations in an I/O-intensive scenario, where each VM ran an I/O-intensive workload based on the well-known `stress` benchmark [68] for a duration of 60 seconds. Overhead samples were collected using Xen's built-in tracing framework by adding tracepoints around key operations within the scheduler.

Table 1 shows the mean overhead (in μs) of three scheduler operations on our 16-core server: **(i)** the time taken to make a scheduling decision, **(ii)** the time taken to process wake-up interrupts, and **(iii)** the time taken to perform any operations after de-scheduling a vCPU, such as sending re-schedule IPIs to another core.

Table 1: Average runtime overheads (in μs) for three key scheduler-related operations on a 16-core, 2-socket server.

	Credit	Credit2	RTDS	Tableau
Schedule	8.08	3.51	2.86	1.43
Wakeup	2.12	5.19	3.90	1.06
Migrate	0.32	5.55	9.42	0.43

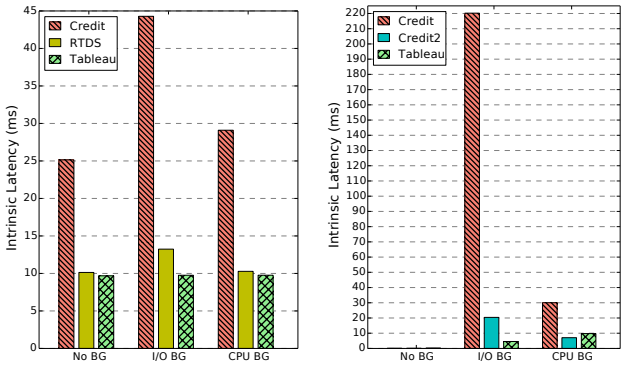
Our focus on runtime efficiency in Tableau's design (Sec. 3) and the optimized, core-local implementation of Tableau's dispatcher (Sec. 6) is clearly reflected in its low scheduler overheads. We observe that Tableau indeed incurs substantially lower overheads compared to other schedulers: the mean scheduling overhead under Tableau is around 5.6x, 2.4x, and 2x lower than under Credit, Credit2, and RTDS, respectively. Concerning post-scheduling operations ("Migrate"), recall that Tableau may occasionally need to send an IPI after de-scheduling a vCPU. As expected, this results in only a negligible increase in the overhead (approximately an additional 100ns on average compared to Credit in our example).

RTDS incurs significantly higher overhead (over 9μs) for post-schedule work due to requiring the acquisition of a global lock when load-balancing vCPUs. To highlight this bottleneck, we also collected overhead data on a 48-core server machine with four sockets (each comprised of 12 cores). Table 2 shows the observed overheads. It is obvious that RTDS' global lock does not scale well: on average, RTDS spends *over 168μs while attempting to migrate a VM* each time it is preempted. We do not consider this machine any further in the remainder of this section.

Table 2: Average runtime overheads (in μs) for three key scheduler-related operations on a 48-core, 4-socket server.

	Credit	Credit2	RTDS	Tableau
Schedule	16.40	4.70	4.39	2.49
Wakeup	7.07	5.61	19.16	1.82
Migrate	0.42	18.19	168.62	0.66

Finally, Table 1 shows the mean overhead for processing wakeups to be 2× lower than under Credit, almost 5× lower compared to Credit2, and over 3× lower than under RTDS. This is a consequence of Tableau's fast wakeup handling (Sec. 6), which uses the table to determine which CPU to send an IPI to.



(a) Capped VMs (b) Uncapped VMs

Figure 5: Maximum scheduling delay as measured by `redis-cli`. "BG" denotes background workload.

To summarize, the advantages of Tableau's design choices are reflected in its efficient runtime compared to other schedulers. This is the result of moving VM budget and latency enforcement to an offline planner, using per-core data structures, and the use of a minimal table-driven dispatcher.

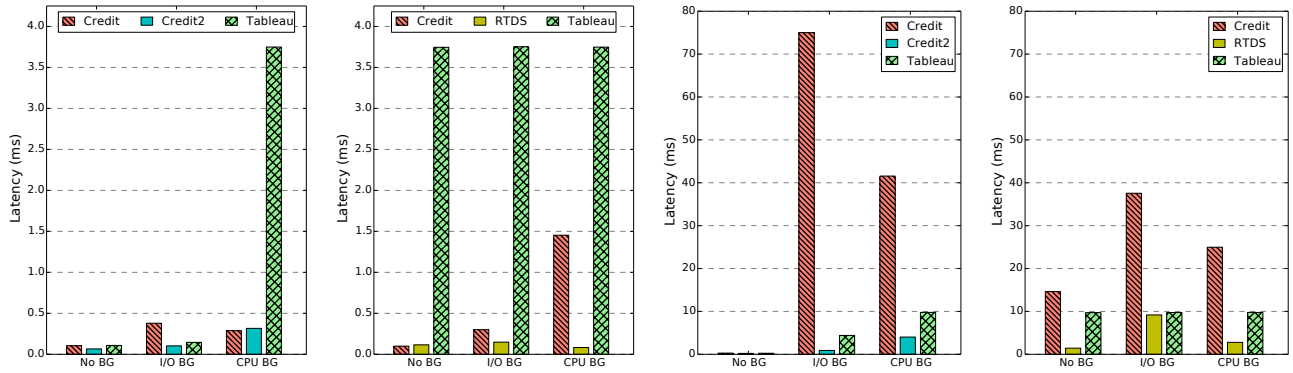
7.3 Comparing Scheduling Delay

To understand the scheduling delays induced by the existing Xen schedulers and Tableau, we used **(i)** the popular `redis-cli` workload with the `--intrinsic-latency` option, and **(ii)** measured the ping latency between our client machine and one of the VMs. The two workloads were chosen as they allow insight into the performance of respectively CPU-bound and sporadically activated, network-I/O-centric VMs under each scheduler. In the following, we present measurements from a single *vantage VM*. The vantage VM did not receive any special treatment or configuration advantages and is thus representative of general scheduler performance.

`redis-cli` intrinsic latency. `redis-cli` is a command-line interface distributed as part of the `redis` key-value store. We ran it within our vantage VM and measured the *intrinsic latency* of the system. When measuring the intrinsic latency, `redis-cli` runs a tight CPU-bound loop and measures the delay between iterations, thus measuring if any delays occur due to the scheduler.

To isolate the effect of the VM scheduler, we ran the tool with the highest SCHED_FIFO priority to avoid interference arising from the Linux scheduler in the guest VM. We evaluated both capped and uncapped scenarios, with four VMs per core, without any background workload, with an I/O-intensive background workload, and with a CPU-intensive background workload. The results are illustrated in Figs. 5(a) and 5(b).

In the capped scenario shown in Fig. 5(a), regardless of the background workload, the scheduler causes scheduling delays as it forcibly cuts off CPU access to VMs once they exceed their assigned amount. In the case of Credit, the VM experiences delays of up to almost 44 ms. Under RTDS, configured as discussed in Sec. 7.2, this results in around 10 ms in the best case with no background workload (*i.e.*, the VM runs at the beginning of each period); more



(a) Uncapped VMs, Average (b) Capped VMs, Average (c) Uncapped VMs, Maximum (d) Capped VMs, Maximum

Figure 6: Average and maximum-observed round-trip ping latencies. "BG" denotes background workload.

latency (up to 13ms) was observed in the presence of a background workload. Finally, under Tableau, we always see about 10 ms of scheduling delay, regardless of background workload. In this experiment, RTDS controls scheduling latency just as well as Tableau, but we will later show that it does not achieve the same throughput.

In the uncapped scenario shown in Fig. 5(b), VMs are not rate-limited and are allowed to consume additional idle cycles if available. As a result, when no background workload is present, all schedulers achieve sub-millisecond scheduling latencies, and the corresponding bars are barely visible in Fig. 5(b). However, latency becomes substantially worse under Credit and Credit2 as a background workload is introduced. In this case, the responsibility of maintaining low scheduling latency for all VMs falls on the scheduler, and as can be seen, it does not work well in high-density scenarios: we observed delays as high as 220 ms under Credit. Credit2 fares well in the presence of a CPU-intensive background workload, but not so well in the presence of the I/O-intensive workload. In contrast, under Tableau, the burden of meeting scheduling latency bounds is the responsibility of the semi-offline planner, which is oblivious to background workloads. As a result, Tableau exhibits at most 10 ms of scheduling delay regardless of background workload.

Ping latency. To cross-validate our findings, we also measured the average and maximum observed ping latency from our client machine to the vantage VM. ICMP echo requests are handled directly within the guest kernel, which eliminates any dependence on the guest scheduler, but can only be processed when the VM is dispatched by the VM scheduler. As a result, with a controlled network like in our setup, the ping latency is dominated by (and is a good proxy for) the scheduling latency incurred by a VM in reaction to wake-ups triggered by external I/O events.

We again evaluated both capped and uncapped scenarios, without any background workload, with an I/O-intensive background workload, and with a CPU-intensive background workload. The experiment setup consisted of eight threads on our client machine, each sending 5,000 randomly-spaced pings with delays ranging from zero to 200 ms. The resulting 40,000 samples were aggregated to determine the average and the maximum observed ping latency for each configuration. The results are reported in Figs. 6(a)–6(d).

In the uncapped scenario, without a background workload, the average latency (Fig. 6(a)) is low for all schedulers (around 100 μs) as the VM can always react immediately to incoming packets. In contrast, the capped scenario (Fig. 6(b)) shows the impact of the table's rigid structure, which results in Tableau exhibiting clearly higher average latency (but well below the configured latency goal of 20 ms).

With an I/O workload in the uncapped scenario, since background VMs frequently block, the vantage VM is able to leverage the resulting idle cycles to achieve a low average latency (Fig. 6(a)). In the case of a CPU-bound background workload, however, there are no additional idle cycles to be had and the vantage VM can only execute during its own slots, which are active only periodically. Thus, the average latency under Tableau is noticeably higher (Fig. 6(a)), but still well below the configured latency goal, since it is determined by the gaps between slots in the table. In contrast, under the other schedulers, which are dynamic in nature and employ heuristics that favor I/O workloads, the vantage VM is able to (on average) respond almost immediately since it is allowed to preempt the predominantly CPU-bound background VMs. However, as we show in Sec. 7.4, these same features can also reduce application throughput and lead to increased unpredictability.

In the uncapped scenario, the maximum observed latencies in an otherwise idle system are around 200 μs (Fig. 6(c)). However, once a background workload is introduced, the maximum observed latency increases under all schedulers. Under Credit, we observe latencies approaching 75 ms in the presence of an I/O-intensive background workload (Fig. 6(c)). Credit2 continues to provide good tail latency characteristics, but as we will show in Sec. 7.4, it is unable to maintain high throughput in this scenario.

In the capped scenario (Fig. 6(d)), the maximum observed latencies under Credit are significantly higher even without any background workload. This is simply because, while VMs are not running any benchmark, they still require CPU time occasionally for system processes. As a result the vantage VM may, under rare circumstances, exhaust its budget, while simultaneously having to wait for the other three background VMs on the same core to exhaust their budget, resulting in up to 15 ms of scheduling latency. While in principle RTDS is also susceptible to the same worst-case

behavior, the necessary conditions did not trigger during our experiment because they occur only very rarely.

With an I/O background workload active, Credit exhibits tail latencies of around 30 ms (Fig. 6(d)). On the other hand, RTDS and Tableau enable accurate control over the scheduling delay. The maximum observed ping latency under RTDS is around 9 ms (Fig. 6(d)), somewhat less than the delay allowed in each period. Similarly, regardless of the background workload, Tableau never exhibits latencies above 10 ms (Fig. 6(d)), which reflects the structure of the table that the planner created for this workload.

To summarize, Credit shows substantially increased tail latency under load in a high-density scenario. While Credit2 and RTDS show good latency characteristics, they struggle to do so while maintaining high throughput, as we show next.

7.4 Comparing `nginx` HTTPS Throughput

We now present a comparison of Tableau, RTDS, Credit, and Credit2 in terms of their respective impact on *application* throughput and latency, as exemplified by the `ngingx` web server.

We used `wrk2` [2], an extension of the well-known `wrk` HTTP load generation tool, that allows for accurate measurement of tail latencies while accounting for the *Coordinated Omission* problem [66].

Setup. Our setup again comprised of four single-core VMs per core on twelve cores of our two-socket, 16-core server (*i.e.*, a total of 48 VMs), with the remaining four cores being dedicated to dom0. Each VM was assigned a virtual network interface using Intel's SR-IOV technology that allowed it to bypass the I/O scheduler in dom0.

The vantage VM was hosting an `nginx` server that served a small PHP "application" via HTTPS. The PHP application simply sends a randomly selected file of a given size (1 KiB, 100 KiB, or 1 MiB) chosen from a 1 GiB dataset. To minimize measurement noise, all files were stored in `tmpfs`. Similarly, `nginx` was assigned a real-time priority to take the guest OS's scheduler out of the picture.

The client machine hosted the `wrk2` tool, which generated requests for a specific file size (1 KiB, 100 KiB, or 1 MiB) at a given rate, and measured the achieved throughput and latency characteristics of the requests. We increased the request rate progressively until the server was saturated. As in the previous experiments, we evaluated both capped and uncapped scenarios, with and without an I/O-intensive background workload.

Graphs. The results are illustrated in Fig. 7, comprising three columns and six rows. The first three rows (Fig. 7 (a)–(i)) show to the capped scenario and the last three rows (Fig. 7 (j)–(r)) show the uncapped scenario. Each row corresponds to the results for either 1 KiB files, 100 KiB files, or 1 MiB files. Within each row, the three columns correspond to the mean, 99$^{\text{th}}$ percentile, and the maximum observed latency, respectively, versus the observed throughput.

Each graph comprises three curves for Credit, Tableau, and either RTDS (for capped scenarios) or Credit2 (for uncapped scenarios). The X-axis shows the observed throughput, while the Y-axis shows a latency metric. Thus, lower is better (*i.e.*, less latency), as is being further to the right (*i.e.*, higher throughput). At some point, the server can no longer keep up with the request rate, and the curve peaks upwards as queueing delays start to dominate.

Capped VMs (Fig. 7 (a)–(i)). In the following, we discuss results for 1 KiB and 100 KiB files (the first two rows); we revisit Tableau's performance with 1 MiB files (Fig. 7 (g)–(i)) later in Sec. 7.5. We make the following key observations.

Tableau provides good tail latencies. The 99$^{\text{th}}$ percentile and the maximum observed latency under Tableau are lower than under Credit and RTDS (Fig. 7 (b)–(c) and Fig. 7 (e)–(f)). While for low request rates, Credit and RTDS's tail latencies are sometimes on par with Tableau's, they quickly increase with the request rate. In contrast, Tableau continues to maintain relatively stable tail-latency characteristics until the server reaches its peak throughput.

Tableau supports higher SLA-aware peak throughput. In both the 1 KiB and 100 KiB scenarios, Tableau achieves a higher peak throughput. In addition, Tableau's latency begins to creep upwards much later than under Credit and RTDS. Thus, given a latency-based *service-level agreement* (SLA), Tableau supports a higher SLA-aware throughput. For example, for 1KiB files, given an SLA that mandates a 99th-percentile latency of 100ms or lower, the peak throughput for RTDS and Credit is around 1,000 and 1,400 requests per second, respectively (see Fig. 7 (b)). Tableau can support up to 1,600 requests per second while satisfying the SLA.

Tableau's rigidity affects its mean latency. The mean latency (Fig. 7 (a) and (d)) under Tableau is higher than under either Credit or RTDS for low request rates. This is expected in a table-driven scheduler, as a request arriving just after the end of a VM's slot has to wait until the next slot of the VM to be processed, while dynamic schedulers like Credit and RTDS can react to the request immediately. However, both schedulers become overwhelmed as the request rate increases, while Tableau's rigidity becomes advantageous and translates into stability at higher request rates.

RTDS struggles to sustain high throughput. In the presence of an intense I/O background workload that causes frequent scheduler invocations, RTDS achieves significantly lower peak throughput than either Credit or Tableau. This is apparent for all three file sizes, and highlights that high VM scheduling overheads can substantially reduce guest application performance.

Credit is significantly less predictable. Mean, 99$^{\text{th}}$, and maximum observed latencies under Credit start to increase significantly before peak throughput is reached. For instance, in graphs (b), (c), (e), and (f) of Fig. 7, Credit exhibits a noticeable upwards slope before peaking (note the log scale), which reflects upon increasing unpredictability as the system becomes increasingly busy. This supports the observation that Credit's I/O boosting heurstic can backfire when faced with I/O-intensive workloads.

Uncapped VMs (Fig. 7 (j)–(r)). Recall that, in the uncapped scenario, Tableau allows each VM to execute in any idle time available on its core, with multiple contending VMs being allocated idle time in a round-robin manner. The challenge for the scheduler is thus to ensure that interference and overheads do not consume precious CPU cycles, thereby degrading the performance of the system. This is where Tableau's low-overhead, table-driven design shines: it maintains stable latency characteristics for significantly higher throughputs compared with Credit and Credit2 for all file sizes. We detail our observations in the following.

Tableau supports significantly higher throughput. In all cases, Credit's performance starts to degrade already at a very low throughput. While Credit2 performs well at low throughput, the peak throughput achieved under Credit2 is still considerably less than

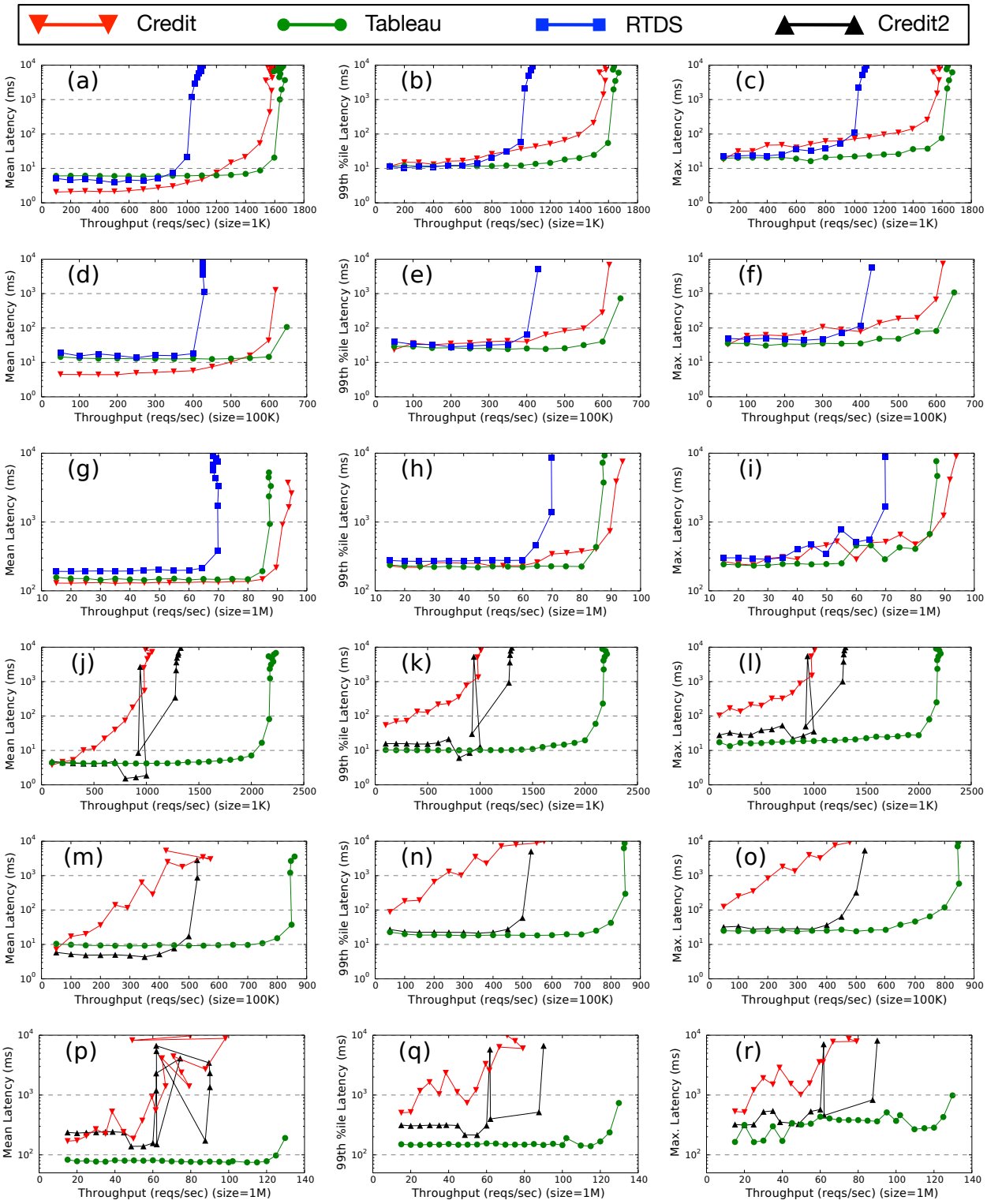


Figure 7: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for capped (first three rows) and uncapped scenarios (last three rows), for 1 KiB, 100 KiB, and 1 MiB files (see y-axis labels), with an I/O-intensive background workload and with varying throughput.

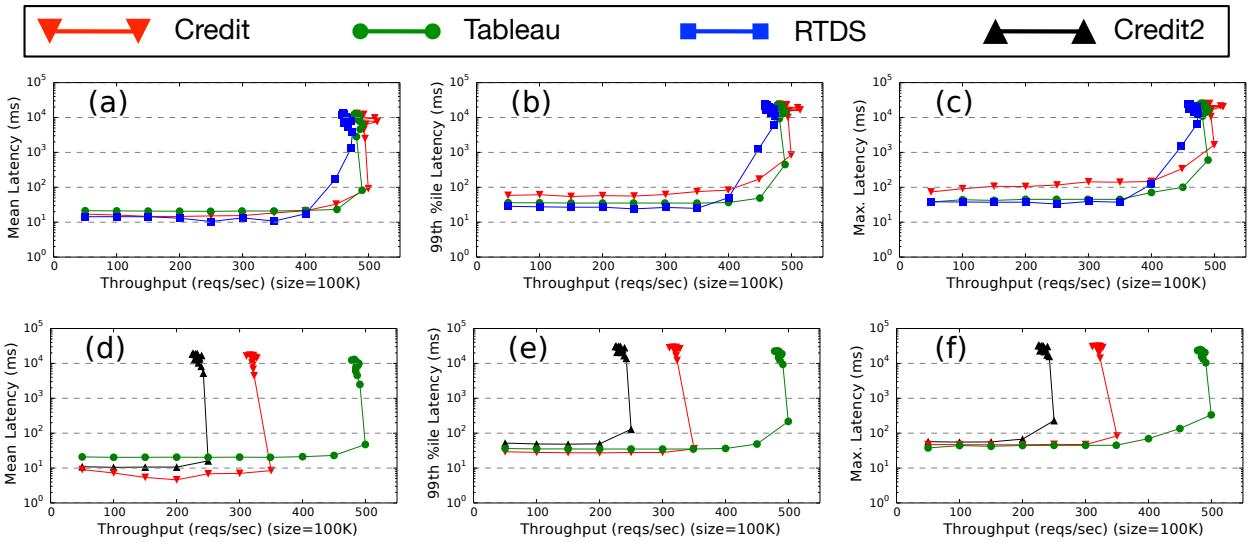


Figure 8: Mean (first column), 99th percentile (second column), and maximum (third column) observed latency for capped (first row) and uncapped VMs (second row) for 100 KiB files with a cache-thrashing background workload and varying throughput.

the peak throughput achieved under Tableau. For example, with 100 KiB files and a 99th-percentile SLA of 100 ms (Fig. 7 (n)), Credit supports only 50 requests per second, Credit2 supports up to 500 requests per second, but Tableau is able to support more than 800 requests per second, about 60% more than Credit2.

Tableau's second-level scheduler is effective. From Fig. 7 (n) and Fig. 7 (e), we can see that Tableau achieves a higher peak throughput of around 850 requests per second in an uncapped scenario compared with around 600 requests per second in the capped scenario. This is due to Tableau's second-level scheduler, which allows the vantage VM to use any idle cycles in the system in addition to its pre-determined slots. To evaluate the contributions of the second-level scheduler, we traced Tableau's scheduling decisions while fixing the request rate at 700 requests per second (supported by Tableau only in the uncapped scenario). We observed that over 85% of the scheduling decisions resulting in the vantage VM's execution were made by the level-2 round-robin scheduler. That is, idle cycles are efficiently and opportunistically allocated by Tableau to other VMs on the same core, which translates into improved throughput without a significant latency penalty.

Cache-thrashing background workload. We now contrast how the schedulers perform in the presence of `stress`'s cache-thrashing background workload, which is fully CPU-bound. Fig. 8 shows the results for 100 KiB files; one row for the capped scenario (Fig. 8 (a)–(c)) and one row for the uncapped scenario (Fig. 8 (d)–(f)). Results for other file sizes were similar; we summarize the main trends.

All schedulers perform similarly in the capped scenario. Since the background workload now is fully CPU-bound—none of the cache-thrashing background VMs ever voluntarily triggers the VM scheduler—the rate of scheduler invocations, and hence the impact of scheduling overheads, is much reduced. As a result RTDS fares much better, and can perform more or less as well as the other schedulers. Fundamentally, Fig. 8 (a)–(c) show a case where the VM scheduler is hardly a bottleneck, and hence it is not surprising to see little differentiation among the schedulers.

Credit outperforms Credit2 due to effective boosting in the uncapped scenario. Credit's boosting heuristic was ineffective in the prior experiment (Fig. 7) since all VMs were I/O-bound and thus all (or effectively none) were prioritized. However, with a cache-thrashing background workload, Credit's boosting heuristic works as intended and plays in favor of the vantage VM, which is the sole VM performing I/O. On the other hand, Credit2, which does not explicitly favor I/O workloads, achieves a lower peak throughput, as can be seen in Fig. 8 (d)–Fig. 8 (f).

Finally, *Tableau outperforms both Credit and Credit2 in the uncapped scenario.* This is where Tableau's rigid table-driven design works best compared to Credit and Credit2's dynamic, heuristic-based designs, which struggle to maintain fairness given the aggressive CPU demand of the uncapped background workload. When comparing the peak throughput under Tableau in the capped and uncapped scenarios (first row vs. second row), we see no drop in Tableau's peak throughput (around 500 requests per second in both cases) as the vantage VM is guaranteed its utilization in both cases, while both Credit and Credit2 see a significant reduction in throughput due to increased interference from uncapped background VMs.

This experiment nicely demonstrates Tableau's advantage in ensuring that each VM receives its guaranteed minimum amount of service no matter what the rest of the system is doing.

7.5 Discussion and Limitations

A rigid table-driven scheduler like Tableau is not ideal for certain scenarios. We next discuss some of the limitations of Tableau.

Lower I/O device utilization in certain capped scenarios. One of the drawbacks of a table-driven scheduler is that I/O requests are sent in periodic bursts. For example, when a VM's slot is active, it is able to enqueue packets in the network interface's ring buffer, but

when the VM is preempted for a relatively long time, the network device drains its buffer and then idles. This is inefficient and results in lower throughput than a dynamic scheduler, which can ensure a more even distribution of VM execution over time, resulting in a better-utilized I/O device given the same CPU utilization.

This effect is evident in Fig. 7 (g)–(i), which shows a capped scenario with 1 MiB files where Credit achieves higher peak throughput compared to Tableau. This does not occur for smaller file sizes as they require less bandwidth (*i.e.*, utilizing the network efficiently is not so important because CPU utilization is the bottleneck when serving small files). However, for larger files, transmission time becomes significant and the VM must work harder to keep the network device sufficiently busy to ensure high throughput.

Overall, if the goal is to maximize I/O device utilization, a rigid table-driven scheduler is not ideal. However, Tableau's second-level scheduler for uncapped VMs can help with overcoming this inefficiency, as is evident in Fig. 7 (p)–(r).

Higher mean latencies in capped settings. While Tableau provides high throughput and predictable tail-latency characteristics, it can be seen in Fig. 7 (a), (d), and (g) that it performs worse compared to Credit in terms of mean latency in capped settings. This is not unexpected since capped VMs under Tableau do not have the luxury of responding to requests at any point in time; rather, they are limited to carefully controlled windows of time where they are allowed to process requests. This means that requests arriving in the blackout period between slots incur higher latencies, resulting in increased average latency. However, as can be seen in the other graphs in Fig. 7, dynamic schedulers come with their own tradeoffs, namely lower throughput in the face of frequent scheduler invocations due to more complex, high-overhead scheduling logic.

Other sources of unpredictability. While we focus on one source of performance variability, CPU scheduling, it is merely one piece of the predictability puzzle. Modern server-class machines have many other sources of such performance variability, including shared-cache interference, queueing delays in shared I/O schedulers, or limitations in guest OSs. Case in point, in the experiments discussed in Sec. 7.4 we specifically removed the guest OS scheduler and disk I/O from consideration to minimize measurement noise. A complete system that comprehensively addresses all such issues is beyond the scope of this paper, but an important challenge for future work.

Semi-partitioned VM performance. In this paper, we focus on the (common) case of fully partitioned vCPUs and do not evaluate migrating vCPUs, that is, VMs that are forced to frequently migrate across multiple cores due to semi-partitioning or localized optimal scheduling (or other table generation methods). To reiterate, we consider semi-partitioning to be rare in a controlled cloud setting as operators can pick vCPU utilizations that are easy to partition.

However, in the rare cases where semi-partitioning is unavoidable, there is undoubtedly a performance penalty to be paid by frequently migrating VMs. While Credit, Credit2, and RTDS also frequently migrate vCPUs, there is a significant difference: under these schedulers, *all* vCPUs are (non-deterministically) subject to occasional migration, so the performance penalty evens out over time. In contrast, in Tableau, migration costs are borne exclusively by vCPUs with allocations on multiple pCPUs, which is unfair.

There are several ways around this imbalance. For one, any split vCPU could be "compensated" for the increased overheads by increasing its utilization by a few percentage points and factoring this added resource usage into the cost. Alternatively, one could periodically re-generate the scheduling table to make sure that all vCPUs take a turn being split across cores. It will be interesting to explore the involved tradeoffs in more detail in future work.

8 RELATED WORK

The literature on virtualization is too vast for a comprehensive review. With the advent of cloud computing technologies, even the specific problem of vCPU scheduling has received significant attention in the past decade. In the following, we primarily focus on work that targets similar goals as Tableau, *i.e.*, predictable performance and/or high-density workloads. In addition, we discuss in brief techniques that solve complementary problems, but which can be incorporated into Tableau's table-driven design, as well as research that highlights limitations of Tableau.

Real-time multi-core VM scheduling. The use of hypervisors in real-time systems is common to support legacy applications, and there exists much prior work on real-time multi-core VM scheduling targeting such workloads. For example, to support applications with QoS and soft real-time requirements (*e.g.*, telephony, audio processing), Lee *et al.* [38], Chen *et al.* [15], and Cheng *et al.* [16] modify the Xen scheduler to expose control over scheduling latency directly to VMs, similar to Tableau. However, their work does not deal with multi-tenant cloud environments with high VM density. Hard real-time schedulers for virtualization platforms, such as the work of Lee *et al.* [36], primarily focus on ensuring *a priori* guarantees for applications to meet their deadlines [44], and are willing to sacrifice throughput to reach this goal. The RTDS scheduler in Xen also derives from real-time VM scheduling and was developed in the RT-Xen project [74, 75]. It bounds scheduling latency while focusing on improving VM throughput, but unfortunately, as shown in Sec. 7, it is unable to maintain performance under high-density scenarios with frequent scheduler invocations, and does not scale well to large server machines. In contrast to the aforementioned works, while Tableau uses techniques from hard real-time scheduling theory, it is squarely aimed at virtualization for data center environments. Other works that apply real-time scheduling theory to schedule cloud VMs include proposals by Cucinotta *et al.* [20] and Lin and Dinda [40], however, these approaches differ distinctly in their designs from Tableau, which uses table-driven scheduling.

Evaluating hypervisor schedulers. The deficiencies of popular VM schedulers, when it comes to the performance of latency-sensitive applications, have been well studied [18, 51, 77] with a lot of emphasis on tuning the various parameters available to improve the performance of particular workloads [14, 37]. While we present a more recent evaluation of some of the tradeoffs of Xen's existing VM schedulers, specifically in high-density environments, an exhaustive evaluation of the available parameter space in these schedulers is beyond the scope of this work. Rather, we employ default parameter settings and incorporate well-known best practices, as would typically be the case in real-world deployments.

I/O responsiveness. I/O latency is an important goal of any VM scheduler as I/O workloads are the dominant type of workloads

deployed in public clouds. There has been significant work on changing existing schedulers to improve the responsiveness of such applications [29, 78]. A small sampling of these include works that use dynamic profiling of applications to improve performance [61], that modify Credit's boosting mechanism [26], tweak the scheduler timeslice [76], as well as partition applications based on performance profiles in order to better isolate them against interference [21]. However, in high-density scenarios, such heuristic- or profiling-driven scheduler enhancements come with the risk of difficult to predict, potentially large runtime overheads. In contrast, Tableau provides delay guarantees with little runtime overhead.

Most closely related to our work on Tableau is SageShift [65], which has similar aims as Tableau: improving application latencies while increasing resource utilization. SageShift does this through a strict admission-control component to determine whether a new VM's SLA can be met. Once admitted, the VM's SLA is guaranteed via a VM scheduler that dynamically adjusts, at runtime, both the utilization and scheduling delay of the VM. While a comparison with SageShift would be interesting, we were unable to locate any publicly available source code for the project.

Complementary work. Due to its separate planning phase, there are several areas of work whose techniques may be used to further improve Tableau's performance. In this paper, we presented a simple approach to generating performant tables but this can be extended further, including NUMA-aware scheduling techniques [57, 72], intelligent co-scheduling of VMs for parallel applications [58], and lock-aware scheduling techniques [71, 79].

Adaptive techniques, instead of modifying the scheduler itself, dynamically reconfigure VMs based on their runtime behavior. Various techniques have been presented using (for instance) control theory [53] and machine learning techniques [14]. Another example is DeepDive [50], which identifies and mitigates inter-VM interference by controlling their placement. Similar adaptive techniques can be used with Tableau to periodically optimize scheduling tables. In fact, Tableau enables the use of high-level languages and mature libraries to rapidly prototype such techniques.

Recent work on LightVM [47], a redesign of Xen's control plane, showed that the majority of VM creation, teardown, and reconfiguration overheads in Xen results from performance-unfriendly design choices and that these operations can be sped up significantly. LightVM is complementary to Tableau and combining them would yield an immediate improvement in VM reconfiguration times; however, this will require a more optimized implementation of Tableau's planner (see Sec. 7.1) to match the improvements realized by LightVM, as otherwise the cost of replanning the schedule could become a dominant bottleneck. However, we believe there is indeed much room for improving Tableau's prototype planner.

Finally, Tableau's design is hypervisor-agnostic. It could, for example, be realized in NOVA [64], a secure, capability-based microhypervisor that minimizes the trusted computing base of a virtualization environment. Tableau's focus on simplicity and use of a userspace planner and load-balancer are well-aligned with NOVA.

Containers and unikernels. Recently, there has been a shift towards lightweight, container-based compartmentalization [7, 24, 49, 55, 62, 70, 73] and unikernels [34, 46], which enable lightweight, purpose-built "OS-less" VM images.

With regards to lightweight containers, we believe that they do not invalidate Tableau's design—the Tableau approach can be easily applied to schedule containers instead of vCPUs, provided the containers are sufficiently long-running. That is, for systems where the configuration of application images is relatively static, Tableau remains applicable. In particular, combined with container-orchestration tools like Kubernetes [13], Tableau may be used to declaratively specify performance requirements of containers running on a cluster. With regards to unikernels, as they are built to be lightweight and application-specific, combining them with Tableau would provide significantly increased performance predictability.

However, we note that Tableau is not applicable for certain uses of containers and unikernels (*e.g.*, on-demand spawning of containers to service individual requests [45]), as this breaks Tableau's assumption that VM (or container instance) creation and teardown are relatively infrequent events, which is not the case in such scenarios. Regardless, we do not see such techniques completely replacing traditional virtualization in the foreseeable future.

Other interference sources. VM scheduling is clearly not the only source of unpredictability and "long tails" in data centers. For instance, much prior work has dealt with network performance isolation [3, 23, 27, 28, 54, 69]. Similarly, the memory subsystem is a major source of unpredictability and performance degradation. For example, Heracles [42] shows how to jointly consider all these aspects (scheduling, memory isolation, and network isolation), albeit on the basis of Linux's heuristic-driven CFS scheduler. The contribution of Tableau is to remove the VM scheduler from the list of key contributors to performance unpredictability and "long tails."

9 CONCLUSION

In this paper, we presented the design of Tableau, an unorthodox VM-scheduler based on static scheduling tables. Tableau combines a low-overhead, table-driven dispatcher with the on-demand generation of scheduling tables satisfying the utilization and scheduling-latency constraints of all VMs in the system. We presented a way to quickly find such tables by repurposing relevant real-time scheduling theory, and presented the design and implementation of an efficient implementation of Tableau in Xen 4.9. We have evaluated our Tableau prototype and compared it against three other Xen schedulers (Credit, Credit2, and RTDS) in the context of an I/O-intensive workload, and have shown it to outperform the other schedulers in terms of SLA-aware peak throughput,

In summary, Tableau is a novel, predictable, high-throughput VM scheduler designed to ensure that VM scheduling is not a bottleneck in the support for high-density VM workloads. While in this paper we focused on processor scheduling alone, it is important to note that CPU scheduling is merely one source of unpredictability in the modern servers, and in the future, it would be interesting to combine the approach presented here with other memory, disk, and network isolation techniques [42], to enable even higher-density packing of VMs without giving up performance guarantees.

ACKNOWLEDGEMENTS

We would like to thank the anonymous EuroSys reviewers, and especially our shepherd Julia Lawall, for their insightful comments, which greatly helped to improve the paper.

REFERENCES

[1] 2018. SchedCAT: Schedulability test collection and toolkit. https://www.mpi-sws.org/~bbb/projects/schedcat. (2018). Online; accessed 12 March 2018.

[2] 2018. wrk2: A constant throughput, correct latency recording variant of wrk. https://github.com/giltene/wrk2. (2018). Online; accessed 12 March 2018.

[3] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. 2012. Less is More: Trading a Little Bandwidth for Ultra-low Latency in the Data Center. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*.

[4] James H. Anderson, Vasile Bud, and UmaMaheswari C. Devi. 2005. An EDF-based Scheduling Algorithm for Multiprocessor Soft Real-Time Systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS '05)*.

[5] Björn Andersson, Konstantinos Bletsas, and Sanjoy Baruah. 2008. Scheduling Arbitrary-Deadline Sporadic Task Systems on Multiprocessors. In *Proceedings of the 2008 Real-Time Systems Symposium (RTSS '08)*.

[6] B. Andersson and E. Tovar. 2006. Multiprocessor Scheduling with Few Preemptions. In *Embedded and Real-Time Computing Systems and Applications (RTCSA'06)*.

[7] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. 1999. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)*.

[8] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. 1993. Proportionate Progress: A Notion of Fairness in Resource Allocation. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing (STOC '93)*.

[9] K. Bletsas and B. Andersson. 2009. Notional Processors: An Approach for Multiprocessor Scheduling. In *Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'09)*.

[10] Konstantinos Bletsas and Björn Andersson. 2011. Preemption-light Multiprocessor Scheduling of Sporadic Tasks with High Utilisation Bound. *Real-Time Syst.* 47, 4 (July 2011), 319–355.

[11] Björn B Brandenburg and Mahircan Gül. 2016. Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations. In *Proceedings of the 37th IEEE Real-Time Systems Symposium (RTSS'16)*.

[12] A. Burns, R. I. Davis, P. Wang, and F. Zhang. 2012. Partitioned EDF Scheduling for Multiprocessors Using a C=D Task Splitting Scheme. *Real-Time Syst.* 48, 1 (Jan. 2012), 3–33.

[13] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, omega, and kubernetes. *Queue* 14, 1 (2016), 10.

[14] Faruk Caglar, Shashank Shekhar, and Aniruddha Gokhale. 2016. iTune: Engineering the Performance of Xen Hypervisor via Autonomous and Dynamic Scheduler Reconfiguration. *IEEE Transactions on Services Computing* (2016).

[15] Huacai Chen, Hai Jin, Kan Hu, and Minhao Yuan. 2010. Adaptive audio-aware scheduling in Xen virtual environment. *ACS/IEEE International Conference on Computer Systems and Applications (AICCSA)* (2010).

[16] Kun Cheng, Yuebin Bai, Rui Wang, and Yao Ma. 2015. Optimizing Soft Real-Time Scheduling Performance for Virtual Machines with SRT-Xen. *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (2015), 169–178.

[17] Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat. 2007. Comparison of the three CPU schedulers in Xen. *SIGMETRICS Performance Evaluation Review* 35, 2 (2007), 42–51.

[18] Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat. 2007. Comparison of the three CPU schedulers in Xen. *SIGMETRICS Performance Evaluation Review* 35, 2, 42–51.

[19] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *The 26th ACM Symposium on Operating Systems Principles (SOSP'17)*.

[20] Tommaso Cucinotta, Dhaval Giani, Dario Faggioli, and Fabio Checconi. 2010. Providing Performance Guarantees to Virtual Machines Using Real-Time Scheduling. In *European Conference on Parallel Processing (EuroPar'10)*.

[21] Mostafa Dehsangi, Esmail Asyabi, Mohsen Sharifi, and Seyed Vahid Azhari. 2015. cCluster: a core clustering mechanism for workload-aware virtual machine scheduling. In *Proceedings of the 3rd International Conference on Future Internet of Things and Cloud (FiCloud'15)*.

[22] Friedrich Eisenbrand and Thomas Rothvoß. 2010. EDF-schedulability of Synchronous Periodic Task Systems is coNP-hard. In *Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '10)*.

[23] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. 2015. Queues Don't Matter When You Can JUMP Them!. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*.

[24] Poul henning Kamp and Robert N. M. Watson. 2000. Jails: Confining the omnipotent root. In *In Proc. 2nd Intl. SANE Conference*.

[25] Philip Holman and James H. Anderson. 2005. Adapting Pfair Scheduling for Symmetric Multiprocessors. *J. Embedded Comput.* 1, 4 (Dec. 2005), 543–564.

[26] Taegyu Hwang, Kisu Kim, Jeonghwan Lee, Jiman Hong, and Dongwan Shin. 2016. Virtual machine scheduling based on task characteristic. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC'16)*.

[27] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. 2013. *Silo: Predictable Message Completion Time in the Cloud.* Technical Report MSR-TR-2013-95. http://research.microsoft.com/apps/pubs/default.aspx?id=201418

[28] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Changhoon Kim, and Albert Greenberg. 2013. EyeQ: Practical Network Performance Isolation at the Edge. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI'13)*.

[29] W. Jiang, Y. Zhou, Y. Cui, W. Feng, Y. Chen, Y. Shi, and Q. Wu. 2009. CFS Optimizations to KVM Threads on Multi-Core Environment. In *Proceedings of the 15th International Conference on Parallel and Distributed Systems (ICPADS'09)*.

[30] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a Warehouse-scale Computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*.

[31] M. Katevenis, S. Sidiropoulos, and C. Courcoubetis. 1991. Weighted round-robin cell multiplexing in a general-purpose ATM switch chip. *IEEE Journal on Selected Areas in Communications* 9, 8 (1991).

[32] Shinpei Kato and Nobuyuki Yamasaki. 2008. Portioned EDF-based Scheduling on Multiprocessors. In *Proceedings of the 8th ACM International Conference on Embedded Software (EMSOFT '08)*.

[33] S. Kato, N. Yamasaki, and Y. Ishikawa. 2009. Semi-partitioned Scheduling of Sporadic Task Systems on Multiprocessors. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems (ECRTS'09)*.

[34] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. OSv: Optimizing the Operating System for Virtual Machines. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*.

[35] Hermann Kopetz and Günther Bauer. 2003. The time-triggered architecture. *Proc. IEEE* 91, 1 (2003), 112–126.

[36] Jaewoo Lee, Sisu Xi, Sanjian Chen, Linh TX Phan, Chris Gill, Insup Lee, Chenyang Lu, and Oleg Sokolsky. 2012. Realizing compositional scheduling through virtualization. In *Proceedings of the IEEE 18th Real-Time and Embedded Technology and Applications Symposium (RTAS'12)*.

[37] Min Lee, AS Krishnakumar, Parameshwaran Krishnan, Navjot Singh, and Shalini Yajnik. 2010. Xentune: Detecting Xen scheduling bottlenecks for media applications. In *Proceedings of the 2010 IEEE Global Telecommunications Conference (GLOBECOM'10)*.

[38] Min Lee, A. S. Krishnakumar, P. Krishnan, Navjot Singh, and Shalini Yajnik. 2010. Supporting Soft Real-time Tasks in the Xen Hypervisor. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'10)*.

[39] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt. 2010. DP-FAIR: A Simple Model for Understanding Optimal Multiprocessor Scheduling. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS'10)*.

[40] Bin Lin and Peter A. Dinda. 2005. VSched: Mixing Batch And Interactive Virtual Machines Using Periodic Real-time Scheduling. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC'05)*.

[41] C. L. Liu and James W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM* 20, 1 (Jan. 1973), 46–61.

[42] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*.

[43] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. 2016. The Linux scheduler: a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys'16)*.

[44] Ruhui Ma, Fanfu Zhou, Erzhou Zhu, and Haibing Guan. 2013. Performance Tuning Towards a KVM-based Embedded Real-Time Virtualization System. *J. Inf. Sci. Eng.* 29 (2013), 1021–1035.

[45] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie. 2015. Jitsu: Just-in-time Summoning of Unikernels. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*.

[46] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*.

[47] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*.

[48] Ernesto Massa, George Lima, Paul Regnier, Greg Levin, and Scott Brandt. 2014. Optimal and adaptive multiprocessor real-time scheduling: The quasi-partitioning approach. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS'14)*.

[49] Bill McCarty. 2004. *SELinux: NSA's Open Source Security Enhanced Linux*. O'Reilly Media, Inc.

[50] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. 2013. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (USENIX ATC'13)*.

[51] Luis Obispo, Ryan Hnarakis, and Lynne Slivovsky. 2014. In Perfect Xen, a Performance Study of the Emerging.

[52] Chandandeep Singh Pabla. 2009. Completely fair scheduler. *Linux Journal* 2009, 184 (2009), 4.

[53] Pradeep Padala, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, and Kenneth Salem. 2007. Adaptive Control of Virtualized Resources in Utility Computing Environments. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07)*.

[54] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. 2014. Fastpass: A Centralized "Zero-queue" Datacenter Network. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM'14)*.

[55] Daniel Price and Andrew Tucker. 2004. Solaris Zones: Operating System Support for Consolidating Commercial Workloads. In *Proceedings of the 18th USENIX Conference on System Administration (LISA'04)*.

[56] Paul J Prisaznuk. 2008. ARINC 653 role in integrated modular avionics (IMA). In *Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th*. IEEE, 1–E.

[57] Jia Rao, Kun Wang, Xiaobo Zhou, and Cheng-Zhong Xu. 2013. Optimizing virtual machine scheduling in NUMA multicore systems. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA'13)*.

[58] Jia Rao and Xiaobo Zhou. 2014. Towards fair and efficient SMP virtual machine scheduling. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 273–286.

[59] Paul Regnier, George Lima, Ernesto Massa, Greg Levin, and Scott Brandt. 2011. RUN: Optimal multiprocessor real-time scheduling via reduction to uniprocessor. In *Proceedings of the 2011 IEEE 32nd Real-Time Systems Symposium (RTSS)*.

[60] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the SIGOPS European Conference on Computer Systems (EuroSys'13)*.

[61] Myungjoon Shon, Kisu Kim, Hansol Lee, Sung Y Shin, and Jiman Hong. 2017. DACS: dynamic allocation Credit scheduler for virtual machines. In *Proceedings of the Symposium on Applied Computing (SAC'17)*.

[62] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. 2007. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys'07)*.

[63] Anand Srinivasan and James H Anderson. 2002. Optimal rate-based scheduling on multiprocessors. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*. ACM, 189–198.

[64] Udo Steinberg and Bernhard Kauer. 2010. NOVA: A Microhypervisor-based Secure Virtualization Architecture. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys'10)*.

[65] O. Sukwong, A. Sangpetch, and H. S. Kim. 2012. SageShift: Managing SLAs for highly consolidated cloud. In *INFOCOM, 2012 Proceedings IEEE*. 208–216.

[66] Gil Tene. 2013. How not to measure latency. *Low Latency Summit* (2013), 68.

[67] Kenneth van Surksum. 2013. The CPU Scheduler in VMware vSphere 5.1. (2013).

[68] Amos Waterland. 2013. stress POSIX workload generator. http://people.seas.harvard.edu/~apw/stress. (2013).

[69] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. 2011. Better Never Than Late: Meeting Deadlines in Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2011 Conference (SIGCOMM'11)*.

[70] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. 2002. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the 11th USENIX Security Symposium*.

[71] Song Wu, Haibao Chen, Sheng Di, Bing Bing Zhou, Zhenjiang Xie, Hai Jin, and Xuanhua Shi. 2015. Synchronization-Aware Scheduling for Virtual Clusters in Cloud. *IEEE Transactions on Parallel and Distributed Systems* 26 (2015), 2890–2902.

[72] Song Wu, Huahua Sun, Like Zhou, Qingtian Gan, and Hai Jin. 2016. vProbe: Scheduling Virtual Machines on NUMA Systems. In *Proceedings of the 2016 IEEE International Conference on Cluster Computing (CLUSTER'16)*.

[73] Miguel G. Xavier, Marcelo V. Neves, Fabio D. Rossi, Tiago C. Ferreto, Timoteo Lange, and Cesar A. F. De Rose. 2013. Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments. In *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'13)*.

[74] Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. 2011. RT-Xen: Towards real-time hypervisor scheduling in Xen. In *Proceedings of the International Conference on Embedded Software (EMSOFT'11)*.

[75] Sisu Xi, Meng Xu, Chenyang Lu, Linh T. X. Phan, Christopher Gill, Oleg Sokolsky, and Insup Lee. 2014. Real-time Multi-core Virtual Machine Scheduling in Xen. In *Proceedings of the 14th International Conference on Embedded Software (EMSOFT'14)*.

[76] Cong Xu, Sahan Gamage, Pawan N. Rao, Ardalan Kangarlou, Ramana Rao Kompella, and Dongyan Xu. 2012. vSlicer: Latency-aware Virtual Machine Scheduling via Differentiated-frequency CPU Slicing. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing (HPDC'12)*.

[77] Xianghua Xu, Peipei Shan, Jian Wan, and Yucheng Jiang. 2008. Performance Evaluation of the CPU Scheduler in Xen. In *Proceedings of the International Symposium on Information Science and Engineering (ISISE'08)*.

[78] Lingfang Zeng, Yang Wang, Wei Shi, and Dan Feng. 2013. An Improved Xen Credit Scheduler for I/O Latency-Sensitive Applications on Multicores. In *Proceedings of the 2013 International Conference on Cloud Computing and Big Data (CLOUDCOM-ASIA'13)*.

[79] Alin Zhong, Hai Jin, Song Wu, Xuanhua Shi, and Wei Gen. 2012. Optimizing Xen hypervisor by using lock-aware scheduling. In *Cloud and Green Computing (CGC), 2012 Second International Conference on*. IEEE, 31–38.